www.javaspektrum.de

Ausgabe 2/2020
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 02

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Domain-Driven Design – Vom Problemraum zur Lösung

Domain-Driven Design

Vom Problemraum zur Lösung

Interview

Thomas Bönig, CDO des IT-Referats Stadt München, arbeitet an der Erschaffung der digitalen Metropole

Ein Wegweiser im DDD-Dschungel

Hochperformante Teams durch DDD

Fachthemen

Locking in Java – effizient synchronisieren

Pragmatisch zum System mit Google Go

zum Inhalt

G30325

Du kommst hier nicht rein!

Locking in Java

Christian Heitzmann

Dass beim Einsatz parallel laufender Threads kritische Bereiche vor konkurrierenden Zugriffen geschützt werden müssen, ist unter Java-Programmierern hoffentlich bekannt. Weniger bekannt hingegen ist, dass Prozesssynchronisationen oft schon in Situationen notwendig werden, die es auf den ersten Blick gar nicht vermuten lassen. Dieser Artikel beschreibt derartige Fallstricke und zeigt die Grundwerkzeuge moderner Java-APIs auf, mit denen sich effizient synchronisieren lässt.

Als einführendes Beispiel möchte ich eine kleine Rätselaufgabe stellen, welche in Listing 1 in gekürzter Form abgedruckt ist. Sämtliche Quellcodes können in vollständiger Form wie üblich unter [Code] heruntergeladen werden.

Über einen `ExecutorService` werden fix acht Threads erstellt und gestartet, die jeweils die Methode `incrementCounterInLoop` ausführen. Diese wiederum inkrementieren in jeweils 100 Millionen Einzelschritten das gemeinsame Objektattribut `counter` vom Typ `long`. Mit

```
import java.util.concurrent.*;

public final class IncrementingCounter {
  private static final long LOOP_COUNT_MAX = 100_000_000;
  private static final int THREAD_COUNT = 8;
  private long counter = 0;

  public static void main(String[] args) {
    IncrementingCounter counter = new IncrementingCounter();
    counter.count();
  }
  public void count() {
    ExecutorService service
      = Executors.newFixedThreadPool(THREAD_COUNT);
    for (int threadNum = 1; threadNum <= THREAD_COUNT; threadNum++) {
      service.execute(() -> incrementCounterInLoop());
    }
    /* [...] Service shutdown omitted. */
    System.out.format("Counter = %,d%n", Long.valueOf(counter));
  }
  private void incrementCounterInLoop() {
    for (long i = 1; i <= LOOP_COUNT_MAX; i++) {
      /* [...] Interrupted check omitted. */
      counter++;
    }
  }
}
```

Listing 1: IncrementingCounter

anderen Worten: Acht Threads erhöhen jeweils 100-Millionen-mal `counter` um 1. Wie groß ist abschließend der Wert von `counter`?

Offenbar kommt es beim gleichzeitigen Zugriff auf `counter` zu sogenannten *Wettlaufsituationen* (englisch: *race conditions*), die eine teilweise gegenseitige Auslöschung zur Folge haben werden. Aber wie viel machen diese aus? Vielleicht bleiben am Schluss noch 750 Millionen oder 700 Millionen übrig? Probieren Sie es aus!

Das effektive Ergebnis schwankt von Ausführung zu Ausführung. Beobachten lassen sich auf meiner Maschine (MacBook Pro) Werte im Bereich zwischen 120 Millionen und 160 Millionen. Das ist meilenweit von den erwarteten Werten 700, 750 oder 800 Millionen entfernt! Wie kommt es dazu?

Zusammengesetzte Aktionen

Der Fallstrick in Listing 1 ist der Befehl `counter++`. Er sieht zwar atomar aus, ist es aber in Wirklichkeit gar nicht. Beim `++`-Operator handelt es sich nämlich um eine sogenannte *zusammengesetzte Aktion* (englisch: *compound action*), die sich hinter den Kulissen im Bytecode sinngemäß aus den folgenden drei Einzelaktionen zusammensetzt:

```
long oldValue = counter;
long newValue = oldValue + 1;
counter = newValue;
```

Entscheidend ist, dass es zwischen diesen drei Zeilen bei paralleler Ausführung jeweils zu Unterbrechungen durch den Scheduler kommen kann. Dramatisch ist insbesondere die letzte Zeile, die im Falle vorheriger Unterbrechungen aufgrund veralteter Werte die Inkrementierungen der anderen sieben Threads durch Überschreiben wieder zunichtemacht.

Compound-Aktionen begegnet man häufiger, als einem lieb ist. Bei obiger Aktion handelte es sich um eine sogenannte *Read-Modify-Write-Aktion*. Bekannt ist den meisten Lesern wohl auch die *Check-then-Act-Aktion*:

```
if (instance == null) {
  instance = new MyObject();
}
return instance;
```

Diese kommt oft zum Einsatz bei der im Ansatz zwar gut gemeinten, in der Praxis aber meist überflüssigen und dazu noch schädlichen „Performance-Optimierung“ durch *Lazy Initialization* [Bloch18]. Dabei wird nämlich oft übersehen, dass es nach der Überprüfung auf `null` in der Verzweigung der ersten Zeile zu einem Threadwechsel durch den Scheduler kommen kann und ein anderer Thread genau dann `instance` mit einem Wert belegt. Kommt der Ursprungsthread anschließend wieder zum Zug, dann wird er `instance` erneut belegen. Für Singletons wäre das fatal! Abhilfe kann hier zum Beispiel die Implementierung von einfachen Singletons mithilfe von Enums bieten [Heitz19].

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung mit Schwerpunkt technischer Implementierungsthemen und Java spezialisiert hat. Er entwickelt seit über 20 Jahren Software und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

Andere zusammengesetzte Aktionen können sein:

- *Put-if-Absent*
- *Remove-if-Equal*
- *Replace-if-Equal*
- Iterationen (implizite oder explizite `hasNext()`-`next()`-Paare)
- Navigationen (nächstes Element in einer bestimmten Reihenfolge finden)
- usw.

Mit Java Version 8 haben zum Beispiel die Methoden `putIfAbsent()` in das `Map`-Interface und `removeIf(Predicate)` in das `Collection`-Interface Einzug erhalten. Threadsichere Implementierungen dieser Datenstrukturen vorausgesetzt, werden so die zusammengesetzten Aktionen nun in jeweils einer einzigen Methode zusammengefasst, ohne dass der Aufrufer noch einmal um separate Synchronisierung dieser besorgt sein muss.

Atomare Datentypen

Wie lässt sich nun das Problem aus dem Einführungsbeispiel beheben? Prinzipiell könnte man direkt mit „grobem Geschütz" auffahren und jeden Aufruf von `count++` mit einem `synchronized`-Block schützen. Viel schöner und auch viel effizienter ist hingegen der Einsatz spezieller threadsicherer Klassen aus dem Paket `java.util.concurrent.atomic`, die atomares Verhalten auf einzelnen Variablen sicherstellen [JAPIAtomic].

Listing 2 zeigt das Programm aus dem Einführungsbeispiel, in dem `counter` vom Typ `long` durch den Typ `AtomicLong` ersetzt wurde. Der `++`-Operator wurde durch den Aufruf der Compound-Methode `incrementAndGet()` ersetzt. Als Ergebnis wird nun der korrekte Wert von 800 000 000 angezeigt, die Ausführungszeit ist aber etwa um den Faktor 40 länger geworden.

`Atomic`-Klassen sind speziell für ihren Zweck optimiert und sollten im realistischen Einsatz einer manuellen Locking-Lösung deutlich überlegen sein. So machen sich die `Atomic`-Klassen unter anderem direkte Hardware-Unterstützung moderner Prozessoren zunutze, deren Befehlssätze unter anderem auch zusammengesetzte (aber atomar ausgeführte) Aktionen beinhalten. Im Vergleich zum manuellen Locking werden Konflikte auch nicht durch Unterbrechen des aktuellen Threads aufgelöst, sondern durch effizientere Maßnahmen, die in realistischen Szenarien keine Unterbrechung des Threads zur Folge haben [Goetz06].

Intrinsisches und explizites Locking

Das *intrinsische Locking* mittels `synchronized` gehört zum Grundlagenwissen eines jeden Java-Programmierers. Konsistent eingesetzt in sämtliche Signaturen von Methoden, die sich potenziell gegenseitig in die Quere kommen könnten, lässt sich so schnell und einfach ein funktionierendes Locking realisieren. Soll nicht ein gesamter Methodenrumpf geschützt werden, sondern nur ein Ausschnitt davon, dann lässt sich stattdessen auch nur dieser Teil des Codes

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public final class IncrementingAtomicCounter {
  private static final long LOOP_COUNT_MAX = 100_000_000;
  private static final int THREAD_COUNT = 8;
  private static AtomicLong counter = new AtomicLong();

  /* [...] */
  private void incrementCounterInLoop() {
    for (long i = 1; i <= LOOP_COUNT_MAX; i++) {
      /* [...] Interrupted check omitted. */
      counter.incrementAndGet();
    }
  }
}
```

Listing 2: IncrementingAtomicCounter

mit einem `synchronized(Object)`-Block umschließen. Der Parameter erlaubt dabei die Angabe eines beliebigen `Object`s als Lock.

Seit Version 5 bietet Java mit dem Paket `java.util.concurrent.locks` auch sogenanntes *explizites Locking* an [JAPILocks]. Die wichtigste Klasse dabei ist wohl `ReentrantLock` [JAPIReentrantLock]. Ihr typischer Einsatz sieht aus wie in Listing 3 dargestellt.

Auffällig und quasi „überlebenswichtig" ist der `try`-(`catch`)-`finally`-Block. Es ist nämlich unabdingbar, den Lock wieder freizugeben, und zwar auch dann, wenn es zu einer Fehlersituation durch eine geworfene Exception kommt. Während `synchronized` den Lock in einer Fehlersituation von selbst wieder freigibt (unter anderem deswegen die Bezeichnung „intrinsisch"), fällt diese Bürde bei den expliziten Locks dem Programmierer zu. Erfahrungsgemäß werden solche „Kleinigkeiten" schnell einmal vergessen.

Explizites Locking mit `ReentrantLock` hat gegenüber intrinsischem Locking mit `synchronized` diverse Vorteile:

- Beim intrinsischen Locking muss ein Thread notfalls ewig warten, ehe es einen Lock bekommt. Es besteht keine Möglichkeit, das Akquirieren eines Locks unverbindlich auszuprobieren, ebenso wenig besteht die Möglichkeit, das Warten auf einen Lock nach einer gewissen Zeitspanne oder von außen durch `Thread#interrupt()` abzubrechen. All dies ist beim expliziten Locking mithilfe der `ReentrantLock`-Methoden `tryLock()`, `tryLock(long timeOut, TimeUnit)` und `lockInterruptibly()` möglich.
- Mit den oben erwähnten Mechanismen lässt sich die Gefahr eines Deadlocks deutlich minimieren, da sich Strategien implementieren lassen, die andere Locks sofort wieder freigeben, sofern mehrere Ressourcen benötigt werden, die aber nicht alle gleichzeitig verfügbar sind. Die Lösung des berühmten Philosophenproblems ist mit expliziten Locks eine Leichtigkeit [WikiPhilosophen].
- Intrinsische `synchronized`-Locks müssen stets in der gleichen Methode freigegeben werden, in der sie akquiriert wurden. Metho-

```
import java.util.concurrent.locks.*;

public class MyClass {
  private final ReentrantLock lock = new ReentrantLock();

  public void myMethod() {
    lock.lock();
    try {
      /* [...] Method body omitted. */
    } finally {
      lock.unlock();
    }
  }
}
```

Listing 3: ReentrantLock

den- oder gar klassenübergreifende Lockingstrategien können so nicht realisiert werden. Mit expliziten `ReentrantLock`s ist dies möglich.
- `ReentrantLock`s lassen sich einen Booleschen *Fairness*-Parameter mitgeben, der den Threads die Locks strikt in der Reihenfolge ihrer Anfrage zuteilt. Ein „Vordrängeln" (englisch: *barging*) oder „Verhungern" (englisch: *starvation*) anderer Threads ist somit nicht mehr möglich.

Ich favorisiere heute ganz klar explizites Locking. Dies nicht einmal, weil `synchronized` per se schlecht wäre, sondern weil es in meinen Augen in Quellcode selten etwas Schlimmeres gibt, als uneinheitliche Vorgehensweisen. Eine Applikation soll also entweder eindeutig und ausschließlich `synchronized` verwenden, oder sonst – wenn auch nur die geringste Chance besteht, dass diese Funktionalität eines Tages einmal nicht mehr ausreichen sollte – von Anfang an auf explizites Locking setzen.

Ein weiterer Grund, der für den konsistenten Einsatz von explizitem Locking spricht, ist die Klasse `ReentrantReadWriteLock` [JAPI-ReadWriteLock]. Sie verwaltet zwei Locks, einen für Lesezugriffe (auf eine zu schützende Datenstruktur) und einen für Schreibzugriffe. Da sich mehrere Lesezugriffe in der Regel nicht gegenseitig stören, können beliebig viele `ReadLock`s akquiriert werden. Sobald aber auch nur ein einziger Schreibzugriff erfolgen soll und hierzu ein `WriteLock` beantragt wird, müssen alle lesenden Threads ihre Lese-Locks freigeben. In der Klasse `ReentrantReadWriteLock` lassen sich dabei diverse Strategien einstellen, wie schnell und mit welcher Fairness dies erfolgen soll. All diese Optionen zu besprechen, würde den Rahmen dieses Artikels sprengen. Stattdessen sei dem Leser die sehr gute API-Dokumentation zu den besagten Klassen empfohlen.

Wieso nicht einfach alles synchronisieren?

Wieso synchronisiert man nicht einfach großzügig (zum Beispiel methodenweise mit `synchronized`) und im Zweifel lieber einmal zu viel als einmal zu wenig? Ließen sich so nicht eine Menge potenzieller Probleme, die bei der nebenläufigen Ausführung von Programmen entstehen können, vom Hals schaffen?

Vom Hörensagen weiß man, dass `synchronized` (und in der Konsequenz wohl auch die expliziten Locks) „teuer" sei; sinngemäß ein „Befehl", der jedes Mal einen konstanten Zeitbedarf hat, wenn er „ausgeführt" wird. Aus diesem Grund darf man „ja nicht unnötig synchronisieren!"

Diese Aussage hat einerseits einen wahren Kern, ist andererseits aber in der Begründung falsch und kann daher zu falschen Schlussfolgerungen führen. „Unnötig" synchronisierte Methoden stellen in einer Einzelthread-Ausführung kein Performance-Hindernis dar, da heutige JVMs intelligent genug sind, um auf unnötige Synchronisation zu verzichten. In dieser Hinsicht schadet es nicht, „einfach mal zur Sicherheit" zu synchronisieren.

Das Problem ist ein anderes und nennt sich auf Englisch *Contention* (deutsch in etwa: Wettstreit, Gerangel). In einem synchronisierten Kontext darf nur ein einzelner Thread ausgeführt werden. Diese Stelle wird nun zum Nadelöhr und performancemäßig zum Engpass, müssen alle parallel laufenden Threads, bis auf einen, nun ihren Dienst unterbrechen und an dieser Stelle warten, bis sie den Lock erhalten. Parallel angedachte Programme werden somit zunehmend zu seriellen Programmen. Sofern man sich durch Parallelisierung einen Performance-Vorteil erhofft hat, wird dieser mit allzu groben synchronisierten Blöcken wieder zunichtegemacht.

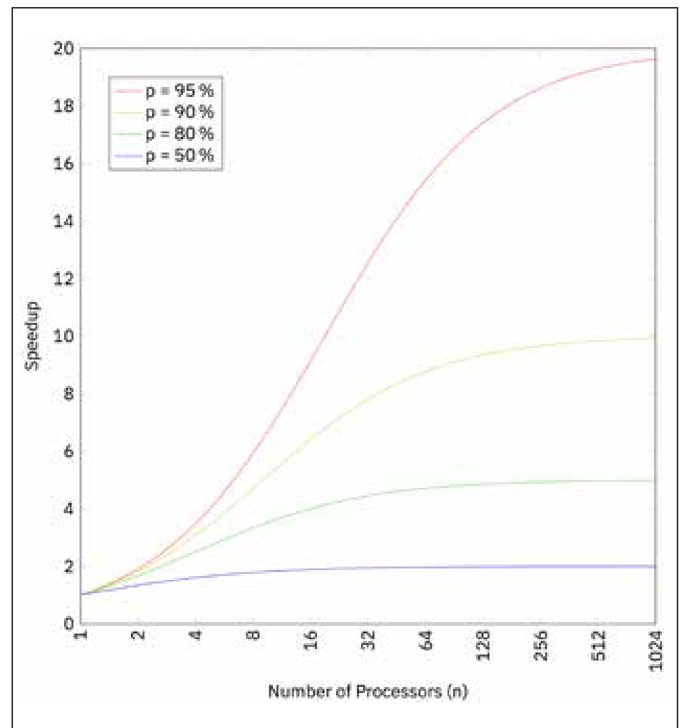

Abb. 1: Amdahlsches Gesetz

Gene Amdahl, ein amerikanischer Computerarchitekt, hat diese Beobachtung bereits 1967 in seinem berühmten *Amdahlschen Gesetz* formalisiert beschrieben [WikiAmdahl]. Die genaue Formel erspare ich dem Leser, verweise dafür auf Abbildung 1, welche diese Formel mit verschiedenen Parametern grafisch wiedergibt. Das Gesetz besagt, dass der zu erwartende Geschwindigkeitszuwachs bei parallelisierten Programmen (sehr) beschränkt ist. Besteht ein Programm zum Beispiel zu 80 Prozent aus parallelisierbarem Code (das heißt, diese 80 Prozent müssen tatsächlich parallel ausführbar sein und dürfen daher auch nicht `synchronized` sein), dann ist auf einer Maschine mit 16 Prozessoren maximal ein 6-facher Geschwindigkeitszuwachs zu erwarten. Interessant ist insbesondere, dass bei 95 Prozent parallelisierbarem Code (das ist sehr sehr viel und außerhalb von wissenschaftlichen Anwendungen wie zum Beispiel Wettersimulationen wahrscheinlich gar nicht zu erreichen) der maximale Geschwindigkeitszuwachs selbst bei über 1000 Prozessoren nicht größer als 20 werden kann.

Zusammenfassend kann gesagt werden, dass synchronisierte Blöcke so klein wie möglich, aber so groß wie nötig gehalten werden sollten. Das ist in der Tat nicht einfach und ein Grund, wieso man von unüberlegtem Multithreading ohne fundiertes Konzept im Zweifel die Finger lassen sollte.

Synchronisierte Datenstrukturen

Wir haben bis jetzt gelernt: Synchronisierung (also die Anzahl der `synchronized`- oder `lock()`-*Aufrufe*) ist per se nicht teuer. Hingegen schlägt die Anzahl der *Code-Zeilen*, die unnötig synchronisiert werden, obwohl sie sicher parallel ausführbar wären, deutlich ins Gewicht.

Ein häufiger Anwendungsfall für Synchronisierung ist der gemeinsame Zugriff auf Datenstrukturen. Listing 4 zeigt den parallelen Zugriff auf eine `SortedSet`, die mittels `Collections.synchronizedSortedSet`-Wrappers für die notwendige Synchronisierung sorgt. Der Datenstruktur werden immer wieder zufällige Werte aus dem Wertebereich eines `Short` hinzugefügt, sodass es zum einen in der Da-

```
import java.util.*;
import java.util.concurrent.*;

public final class ConcurrentCollections {
  private static final long LOOP_COUNT_MAX = 10_000_000;
  private static final int THREAD_COUNT = 8;
  private SortedSet<Short> set
      = Collections.synchronizedSortedSet(new TreeSet<>());
//      = new ConcurrentSkipListSet<>();

  public static void main(String[] args) {
    ConcurrentCollections collections
      = new ConcurrentCollections();
    collections.add();
  }
  public void add() {
    ExecutorService service
        = Executors.newFixedThreadPool(THREAD_COUNT);
    for (int threadNum = 1; threadNum <= THREAD_COUNT;threadNum++) {
      service.execute(() -> addElementsToSet());
    }
    /* [...] Service shutdown omitted. */
  }
  private void addElementsToSet() {
    for (long i = 1; i < LOOP_COUNT_MAX; i++) {
      /* [...] Interrupted check omitted. */
      set.add(Short.valueOf((short) ThreadLocalRandom.current()
                      .nextInt(Short.MAX_VALUE)));
    }
  }
}
```

Listing 4: ConcurrentCollections

tenstruktur „etwas zu tun gibt", namentlich Suchen, Duplikatsprüfungen, Sortierungen und Balancierungen, zum anderen aber die Datenstruktur nicht zu viele Elemente speichern muss und somit der Speicher auch nicht überlaufen wird.

Der Wrapper funktioniert äußerst banal und fügt einfach jeder Methode des `SortedSet`-Interfaces ein `synchronized` hinzu. Bis und mit Java 1.4 war dies die einzige einfache (nicht manuelle) Möglichkeit, die von Haus aus nicht threadsicheren Collections zu synchronisieren.

Lässt man das Programm laufen, so dauert dies auf meiner Maschine 16 Sekunden. Ersetzen wir den groben Wrapper durch eine explizit für den parallelen Einsatz designte Datenstruktur, in diesem Fall einer `ConcurrentSkipListSet`, so reduziert sich die Ausführungszeit bei mir auf 3 Sekunden; ein Geschwindigkeitszuwachs um mehr als Faktor 5!

Die Erklärung ist recht einfach. Wie im vorherigen Abschnitt ausgeführt, gibt es beim parallelen Programmieren aus Performance-Sicht nichts Verheerenderes, als unnötig viele Code-Zeilen zu synchronisieren. Genau das machen aber die Wrapper von `Collections.synchronized...`. Stattdessen kann ich jedem Java-Entwickler nur ans Herz legen, die extra hierfür gedachten threadsicheren Gegenstücke aus dem Paket `java.util.concurrent` einzusetzen [JAPIConcurrent]. Mit deren Design haben sich genügend schlaue Köpfe intensiv beschäftigt, um Threadsicherheit bei bestmöglicher Performance zu realisieren.

Die erwähnten Datenstrukturen, die allesamt mit Java 5 eingeführt wurden, haben leider keine besonders sprechenden Namen, sondern tönen eher kryptisch. Tabelle 1 soll dem Leser den Umstieg erleichtern.

Interface/Klasse	Synchronisierte Klasse
`List/ArrayList`	`CopyOnWriteArrayList`
`Set/HashSet`	`CopyOnWriteArraySet`
`SortedSet/TreeSet`	`ConcurrentSkipListSet`
`Map/HashMap`	`ConcurrentHashMap`
`SortedMap/TreeMap`	`ConcurrentSkipListMap`
`Queue`	diverse

Tabelle 1: Umstieg auf synchronisierte Datenstrukturen

Fazit

Als Grundregel sollte man von paralleler Programmierung absehen, sofern nicht im Voraus der formale Beweis erbracht wurde, dass sich damit signifikante Geschwindigkeitsvorteile erzielen lassen. Ansonsten überwiegen meines Erachtens die Gefahren, wie sie insbesondere bei fehlender oder fehlerhafter Synchronisierung auftreten können. So oder so sind immer zuerst die threadsicheren Datenstrukturen aus dem Java API heranzuziehen, ehe man selber Locking-Mechanismen implementiert.

Auf jeden Fall ist es sehr lohnenswert, sich einmal die API-Dokumentationen zu den erwähnten Paketen anzuschauen. Denn im Wissen, was es alles gibt, wird man umso weniger in Versuchung kommen, es selber noch einmal zu probieren oder nachzubauen.

Literatur und Links

[Bloch18] J. Bloch, Effective Java, 3rd Edition, Item 83: Use lazy initialization judiciously, Pearson Education, 2018

[Code] Quellcode zum Herunterladen, https://link.simplexacode.ch/sacv

[Goetz06] B. Goetz, Java Concurrency in Practice, Pearson Education, 2006

[Heitz19] C. Heitzmann, Der Einsatz von enums abseits reiner Aufzählungen, in: JavaSPEKTRUM, 1/2019

[Hettel16] J. Hettel, M. T. Tran, Nebenläufige Programmierung in Java, dpunkt.verlag, 2016

[JAPIAtomic] Java Platform, Standard Edition 11 API Specification, Package java.util.concurrent.atomic, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

[JAPIConcurrent] Java Platform, Standard Edition 11 API Specification, Package java.util.concurrent, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html

[JAPILocks] Java Platform, Standard Edition 11 API Specification, Package java.util.concurrent.locks, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/package-summary.html

[JAPIReadWriteLock] Java Platform, Standard Edition 11 API Specification, Class ReentrantReadWriteLock, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/ReentrantReadWriteLock.html

[JAPIReentrantLock] Java Platform, Standard Edition 11 API Specification, Class ReentrantLock, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

[WikiAmdahl] Wikipedia, Amdahlsches Gesetz, https://de.wikipedia.org/wiki/Amdahlsches_Gesetz

[WikiPhilosophen] Wikipedia, Philosophenproblem, https://de.wikipedia.org/wiki/Philosophenproblem